Using SystemVerilog for IC Verification

Mikhail Noumerov
Lyubov Zhivova

Motorola GSG-Russia

Mikhail.Noumerov@motorola.com
Lyubov.Zhivova@motorola.com

ABSTRACT

SystemVerilog is an extension to IEEE 1364-2001 "Verilog-2001" standard, which is positioned by EDA vendors as a new unified language for IC design and verification. In addition to SystemVerilog constructs that provide increased engineer productivity through higher level of abstractions, SystemVerilog is intended to improve communication between design and verification teams. SystemVerilog simulation is expected to be faster than simulation of a heterogeneous testbench, i.e. design written in Verilog and VIP written in Vera. In brief, SystemVerilog becomes a promising means of verification for the future use in the semiconductor industry.

Currently there is no full support of SystemVerilog by EDA tools. However, limited syntax is implemented in the VCS compiler and simulator, which are commonly used in Motorola. This paper reports about our trial building of a SystemVerilog testbench, discusses pros and cons and thus considers actual SystemVerilog status and effectiveness of the language utilization at its current state. SystemVerilog comparison with existing technology based on the use of Vera language is also performed in the paper.

Table of Contents

List of Figures

List of Tables

1.0 Introduction

Over the past several years, the growing complexity of electronic system-on-chip devices has made it harder and harder to make designs that are robust, reliable, and bug-free. Many sources give the ratio 20% to 80% of the time for IC development and verification. With growing complexity of SoCs, the same has been going on with testbenches and verification methodology. Starting from pure HDL testbenches and vector sets, IC verification means grew up to transaction based systems which include such complex blocks as bus drivers, interface monitors, reference models, response checkers and so on. Test sequences have become written at a higher level of abstraction. More recently, random sequence generators have come to life. They allow generating and sending transactions to DUV in random order. Transaction based verification methodology gave birth to variety of tools supporting it. C or C++ based libraries and BFMs, TestBuilder from Cadence, VERA from Synopsys – all of them bring opportunity for verification engineers to build effective test environment. However they all have one big minus – they are proprietary tools and they are not based upon any industry standard, as compared to the design world living with either Verilog or VHDL standards, or both of them, but no more. Another minus of those tools is that they all get connected to the HDL world via a PLI interface, which is good, but appears to be a little bit slow in comparison with pure HDL code. Hence that's the price we pay for building complex testbenches with nice tools – these are simulation speed and non-portable code.

An ideal solution for verification engineers would be a language which has the following features:

- easily integrated with DUV;
- supports sequential operation;
- supports multi-threading;
- supports randomization and has a constraint solver;
- supports high level of abstraction and design hierarchy;
- is an industry standard and supported by several EDA vendors.

According to its specification, SystemVerilog seems to be the language that matches these criteria:

- it is a unified language for design and verification – an HDVL;
- it has built-in support for sequential operators and fork – join constructions;
- it provides OOP support and thus is well structured;
- it is an Accellera standard and pushed to IEEE for standardization;
- many EDA vendors already include SystemVerilog support in their products, or plan to do it in the near future.

Synopsys as a major EDA vendor added SystemVerilog support in its VCS design and simulation tool starting from version 7.1, and continues improving SystemVerilog standard language coverage in every new version. Since VCS is a main tool used in Motorola / Freescale Semiconductor for RTL design and simulation, it is important to keep new opportunities on track. This paper is a result of an attempt to get basic knowledge about SystemVerilog, with a goal to evaluate its current status, advantages and disadvantages, and compare this tool with existing technology and verification flow based upon using of Vera language.

2.0 Verification Means of SystemVerilog

SystemVerilog is an extension of the Verilog Hardware Description Language with the higher level of abstraction for modeling and verification. A lot of high level constructs were added into the language and more were derived from C/C++ and Java. Logically these new

constructs can be divided into two groups: constructs dedicated to modeling and constructs dedicated to verification. Let's consider the most important innovations in SystemVerilog which create verification power in the language.

- Program block serves as a clear separator between DUV and the testbench. It specifies execution semantics for all elements declared within the test program. It operates as an entry point for the test program where verification engineer programs the sequence of stimulus and expected response from the DUV. Together with clocking domains, the program block provides race-free interaction between the design and the testbench.

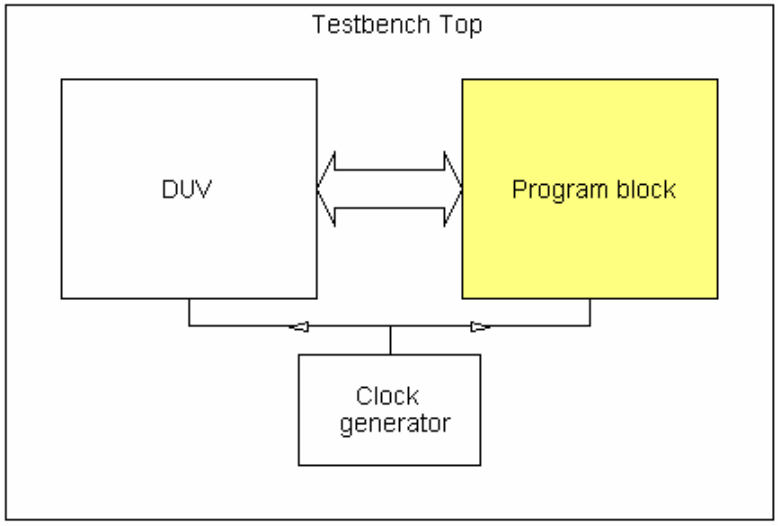

Figure 1. Program Block in a Testbench

- Clocking domain construct identifies clock signals, and captures the timing and synchronization requirements of the blocks being verified. A clocking domain assembles signals that are synchronous to a particular clock, and makes their timing explicit in terms of sampling and driving relative to the clock. This eliminates the risk of having races in the testbench when a signal is sampled and driven at the same moment. Therefore clocking domains play a key role in a cycle-based methodology. Clocking domains separate timing and synchronization details from the structural, functional, and procedural elements of a testbench.

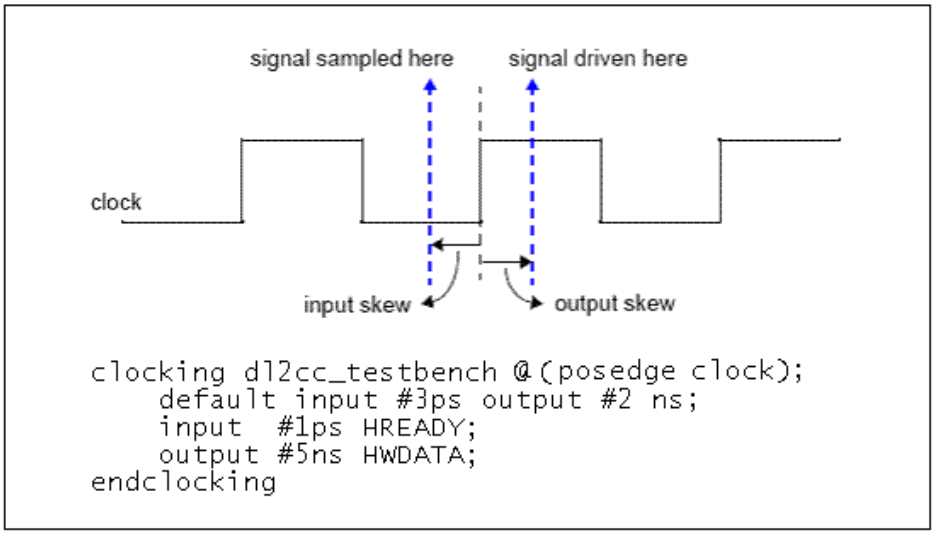

Figure 2. Clocking Domain Example

- Classes and object-oriented programming (OOP) in SystemVerilog offer abstract type modeling, which brings the advantages of the object-oriented paradigm successfully utilized in to software world in C++ and Java languages. The main advantages of the OOP over conventional approaches are:
  - OOP provides a clear modular structure for programs which makes it good for defining abstract data types where implementation details are hidden and the unit has a clearly defined interface.
  - OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
  - OOP provides a good framework for code libraries where supplied software components can be easily adapted and modified by the programmer.

```
class Packet ;
   //data or class properties
   bit [3:0] command;
   bit [40:0] address;
   bit [4:0] master_id;
   integer time_requested;
   integer time_issued;
   integer status;

   // initialization
   function new();
      command = IDLE;
      address = 41'b0;
      master_id = 5'bx;
   endfunction

   // methods
   // public access entry points
   task clean();
      command = 0; address = 0; master_id = 5'bx;
   endtask

   task issue_request( int delay );
      // send request to bus
   endtask

   function integer current_status();
      current_status = status;
   endfunction
endclass
```

Figure 3. Class Definition Example

Classes significantly decrease the dependency of SystemVerilog on the other languages and make it possible to create complex native models without importing C or C++ components in the testbench, thus improving simulation performance, and, as a result, shortening verification cycle time.

- Direct Programming Interface (DPI) makes possible exporting and importing tasks and functions to other languages, such as C or C++. It is not a secret that a lot of models already exist written in C or C++. In order to achieve high level of reuse and thus shortening verification cycle time, using the DPI is encouraged as DPI allows direct inter-language function calls between the languages. The usage of imported functions is identical as for native SystemVerilog functions with the help of DPI. For now, however, SystemVerilog 3.1 defines DPI foreign language layer only for the C programming language, which still covers a lot of existing verification IP components.

- Multithreading is extremely important for verification as usually test programs consist of issuing concurrent sequences of transactions on various DUT interfaces. Fork … join SystemVerilog constructions derived from C and Java are intended to simplify development of concurrency.
- Interfaces allow encapsulating communication protocol signals and behavior in a variety of modes (i.e. master or slave, single beat or burst etc.). They appear as a special kind of classes where signals play the role of internal data structure. Interfaces do not prohibit direct access to the signals. However, the definition and using of the interface methods makes the code more structural, readable and maintainable. Modports are another useful component of interfaces, which enable engineers to define various operational modes of the interfaces, as shown on Figure 4.

```
interface simple_bus (input bit clk); // Define the interface
   logic req, gnt;
   logic [7:0] addr, data;
   logic [1:0] mode;
   logic start, rdy;

   modport slave (input req, addr, mode, start, clk,
                  output gnt, rdy,
                  ref data);

   modport master(input gnt, rdy, clk,
                  output req, addr, mode, start,
                  ref data);

endinterface: simple_bus
```

Figure 4. Interface and Modport Example

Interfaces are assumed to be defined by the DUV designers. However they may also be developed by verification engineers and include protocol checking mechanisms.
Though interfaces are similar to classes, they are missing inheritance and polymorphism features, which would be convenient when interface specification changes. For example, if the ARM AHB specification is extended to AHBv6, it would be more logical to build AHBv6 interface on the basis of AHB by adding or changing signals, methods and modports, rather than to write the AHBv6 interface from scratch.

- Random generator and constraint solver. Random verification is a very important part of the verification strategy, since it allows to:
  - Generate more tests in the same amount of time. Comprehensive verification of today's complex SoCs requires many tests, and random generation of patterns is a viable solution to target this problem.
  - Discover corner case situations and bugs.

  The ability of a programming language to generate random values and sequences bordered by a number of constraints play important role in selection of the language for verification purposes. Certainly general purpose languages have functions for generating random numbers. A big and non-trivial part of the work of verification team though is constraint solver development or reuse. Languages dedicated to verification such as Synopsys Vera or Cadence TestBuilder already have built-in constraint solvers, which can be better or worse in terms of performance, but basically it solves the problem of constrained randomization and lets verification engineers concentrate on their main tasks. SystemVerilog also has a built-in random constraint solver and allows users to declare constraints as shown on Figure 5, which are then processed by the solver that generates random values and sequences.

```
class Bus;
   rand bit[15:0] addr;
   rand bit[31:0] data;

   constraint word_align {addr[1:0] == 2'b0;}
endclass
```

```
Bus bus = new;

repeat (50) begin
   if ( bus.randomize() == 1 )
      $display ("addr = %16h data = %h\n", bus.addr, bus.data);
   else
      $display ("Randomization failed.\n");
end
```

Figure 5. Constraint Declaration Example

- Assertions are expressions specifying the state of a design that must exist at one or more specific points during execution. Assertions can be used to specify the designer's intent and thus improve robustness of the design. The way assertions are used for verification is raising exceptions or fault reports when violated.
  SystemVerilog assertions can be immediate or concurrent. The former evaluate expressions along with the code they are inserted in, and according to the general model simulation rules. The latter provide a mechanism for sampling signal values on clock ticks independently of the simulator, evaluating them with respect to the other signal values and reporting about the error if necessary. Thus concurrent assertions enable checking signal behavior in a non-zero time frame.
  Though SystemVerilog assertions are primarily used to validate the behavior of a model by design engineers, they can also be used by verification teams for interface physical protocol checking. However, assertions are not applicable for packet based protocols verification, where functional coverage measurement helps significantly.
- Coverage measurement. Statement, toggle, FSM, assertion and functional coverage types supported by SystemVerilog provide a means for coverage driven verification. Coverage is based upon counting occurrences of user-specified events, i.e. a pin toggle from 0 to 1 or vice versa, entering certain FSM a state or having all 1's on a data bus. Covergroup construction allows the specification of
  - coverage points for variables, expression and transitions;
  - cross coverage measurement, when aggregated events are counted;
  - sampling expression, which determines the moment for the checking events for occurrence.

  With the definition of the events to monitor and the definition of the coverage goals (number of hit events), the verification engineer is able to have quantitative parameters for measuring probability of the fact the DUV is functional and free of bugs. Together with random test generation, coverage measurement can be used as a driver for extra test pattern generation until desired aggregate coverage is reached in simulation.

3.0 SystemVerilog vs. VERA

As discussed in the previous section, SystemVerilog constructs obviously are not brand new and unique when we compare the language with currently used hardware verification languages or libraries. Moreover existing languages might have an even bigger spectrum of capabilities. Every time a new language is invented, the first questions that developers or verification engineers ask are "why I should learn this language, what advantages does it have, and how will it simplify my work". To answer these important questions, we compared features available to engineers in the Synopsys VERA language, which is now widely used for verification purposes, with equivalent components in SystemVerilog.

Detailed analysis of language specification showed that SystemVerilog and Vera are very similar in language structure, providing the same set of means for verification, including:

- Program block
- Classes & Object Oriented Programming
- Clocking Domains
- Randomization
- Assertions
- Functional Coverage
- Interface with C

However both languages have their own pros and cons. These are summarized in Table 1, below.

Table 1. SystemVerilog and Vera Features Comparison

SystemVerilog	VERA
✓ Unified language for design and verification	× Connected to HDL via PLI interface
✓ Accellera standard and pushed to IEEE for standardization	× Synopsys proprietary
✓ Faster simulation	× Slower simulation
× Less verification constructs	✓ Verification needs dedicated
× Not yet 100% implemented	✓ Available and stable
× No defined methodology	✓ Methodology defined and implemented

The main advantage of SystemVerilog is the possibility to use the same language for design and verification. This results in a good code performance. As shown later, simulation time is about 3 times smaller with SystemVerilog than with Vera. The main disadvantage of the language is that it is not yet fully implemented and supported by known simulators.

In the Vera language, slower simulation time is compensated by the existing of a defined and implemented verification methodology, making Vera more stable and preferred solution from the verification point.

4.0 SystemVerilog Support in Synopsys VCS 7.2

Synopsys VCS is a compiler and simulator, which is commonly used in IC design flow. It is not exceptional in SystemVerilog support on its current stage. SystemVerilog support in VCS is currently modeling side oriented, meaning that constructs are dedicated to reducing the number of lines of code in the design, and improving model code reliability and readability. The constructs include

- Data definition extension including extra data types taken from C:
  - signed/unsigned integers

    - enums
    - typedefs
    - data structures (not unions)
    - multi-dimensional arrays
- Data definition extension including extra data types taken from VHDL:
    - logic
- Improved operator and function definition syntax:
    - operations such as ++ and --
    - void functions
    - outputs and inouts in function ports
    - arrays can be passed as parameters of functions
    - explicit return from tasks and functions
- Important language enhancements:
    - default port connections (which definitely decrease maintainability of the design)
    - interfaces
    - always_comb, always_ff, and always_latch procedural statements
    - fork … join, join_any, or join_none thread control statements.

With a significant focus of the design side support, verification side is left uncovered. Table 2 gives a summary of the important verification constructs described in section 2.0 and their implementation status in VCS 7.2

Table 2. Verification Constructs Supported by VCS

Verification Construct	Implementation Status
Program block	Not supported
Clocking domains	Not supported
Classes and OOP	Not supported
DPI	Not supported
Multithreading	Supported
Interfaces	Supported
Randomization	Not supported
Assertions	Supported
Coverage measurement	Supported all but functional coverage measurement

5.0 Benchmark Testbench

Taking into account the current restrictions for verification constructs, we built a testbench where we tried to use as many new language keywords as possible. The second aim was to do a performance comparison of SystemVerilog and Vera languages not only in theory, but also in practice. The testbench is targeted at verification of a Level 2 Cache Controller and has a minimum set of components for testing basic controller functionality.

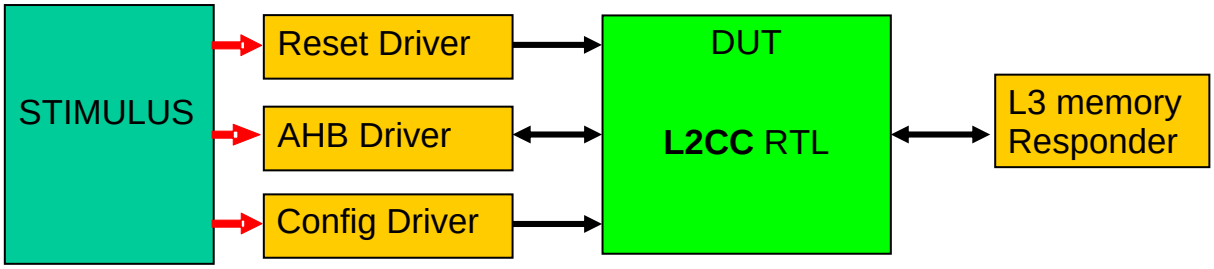

Figure 6. SystemVerilog Evaluation Testbench Structure

The following components compose the testbench (shown in Figure 6):

- DUT (L2 Cache model written on Verilog),
- Drivers
  - Reset driver
  - AHB driver
  - Config driver
- Responder
- Stimulus (test scenario).

5.1 Implementation

It is worth mentioning that we were very limited during development and comparison with VERA due to poor support of SystemVerilog in VCS. For example we were not able to use classes and OOP during development, and were not able to compare means for randomization and functional coverage. SystemVerilog features that we were actually able to try in our investigation are described below in this section.

- Interfaces

Write/Read accesses to L2CC are performed using the AHB protocol. L2CC uses the same protocol to initiate transactions to the L3 memory. The difference is only in width of buses for data, address and control.

The SystemVerilog *interface* construct was very helpful here. The widths of all buses are defined as parameters, which can be changed at the moment of interface instantiation. The AHB bus does not require tri-state or multi-drop interconnect, and therefore it was appropriate to model all the bus signals as vector or scalar logic variables.

```
interface ahb_intf #(parameter HADDR_WIDTH = 32, HDATA_WIDTH = 32,
                              HBURST_WIDTH = 3, HPROT_WIDTH = 4,
                              HRESP_WIDTH = 2, HSEL_WIDTH = 2)
                    (input bit HCLK, HRESETn, bigend);

    // Operation signals
    logic [HADDR_WIDTH-1:0] HADDR;
    TTrans HTRANS;
    logic HWRITE;
    TSize HSIZE;
    logic [HBURST_WIDTH-1:0] HBURST;
    logic [HPROT_WIDTH-1:0] HPROT;
    logic [HDATA_WIDTH-1:0] HWDATA;
    logic [HSEL_WIDTH-1:0] HSEL;
    logic [HDATA_WIDTH-1:0] HRDATA;
    logic HREADY;
    TResp HRESP;
    logic [(HDATA_WIDTH>>3)-1:0] HBSTRB;
    logic HUNALIGN;

    // Arbitration signals
    logic HBUSREQ;
    logic HLOCK;
    logic HGRANT;
    logic [3:0] HMASTER;
    logic [3:0] HDOMAIN;
    logic HMASTLOCK;
    logic [15:0] HSPLIT;

        ...

endinterface : ahb_intf
```

Defined variables can be manipulated by tasks in the interface itself. Such tasks can implement the execution of a bus cycle by some external bus master, in the same manner as a traditional Bus Functional Model (BFM).

```
interface ahb_intf #(parameter HADDR_WIDTH = 32, HDATA_WIDTH = 32,
                              HBURST_WIDTH = 3, HPROT_WIDTH = 4,
                              HRESP_WIDTH = 2, HSEL_WIDTH = 2)
                    (input bit HCLK, HRESETn, bigend);
       ...

           task master_reset;

               while (transfer_in_progress == 1'b1) begin
               end

               HADDR = 0;
               HTRANS = IDLE;
               HWRITE = 0;
               HSIZE = BIT8;
               HBURST = 0;
               HPROT = 0;
               HWDATA = 0;
               HMASTER = 0;
               HMASTLOCK = 0;

               data_queue_size = 0;

           endtask : master_reset


       ...

endinterface : ahb_intf
```

- Modports

Two modports have been used in the testbench.
Master modport is intended for connection to a bus master. Such a master module should directly drive all physical signals in the bus, except the response signals HREADY, HRESP, HRDATA. In our testbench, this modport is used to connect AHB driver which initiates transactions to the cache controller. The test stimulus is expected to perform different read and write cycles by invoking tasks implemented within the AHB interface itself. These tasks are invoked through the modport using *import task*, so that the stimulus does not need to make hierarchical name reference to the interface, but instead can work through its named port.

```
    modport master(
        input   HCLK, HRESETn, HRDATA, HREADY, HRESP, bigend,
        output  HADDR, HTRANS, HWRITE, HSIZE, HBURST, HPROT,
                HWDATA, HMASTER, HMASTLOCK,
        import  master_reset,
                master_write,
                master_read);
```

Slave modport is intended for connection to a bus slave. Bus slaves should receive all physical signals in the bus, except for their response signals. We used slave modport to connect L3 memory responder.

```
  modport slave(
        input   HCLK, HRESETn, HADDR, HTRANS, HWRITE, HSIZE,
                HBURST, HPROT, HWDATA, HSEL, HMASTER, HMASTLOCK, bigend,
        output  HRDATA, HREADY, HRESP,
        import  slave_reset);
```

- Assertions and properties

SystemVerilog provides several constructs which can be used to validate behavior of the design. A *property* defines system behavior and can be used for verification as an

assumption, a checker, or a coverage specification. In our system we used properties to define the expected state of a signal during hard reset, after reset deassertion, and also to define dependencies between some signals.

```
// HSELR should be non X during reset
property p1;
  @(posedge clk) hresetb |-> (hselr !== 1'Bx);
endproperty

...

// HREADYIN should be non X
property p4;
  @(posedge clk) (hreadyin !== 1'Bx);
endproperty

// HSELR shall be asserted together with HSEL only */
property p5;
  @(posedge clk) hselr |-> (hsel == 1'B1);
endproperty
```

Assertions follow simulation event semantics for their execution and are executed like a statement in a procedural block. The *assert* statement is used to enforce a property as a checker. When the property for the assert statement is evaluated to be true, the pass statements of the action block are executed. Otherwise, the fail statement is executed.
We used assertions to check the expected behavior specified by properties on each simulation cycle. In cases when a particular check failed, corresponding error message was displayed.

```
a1: assert property(p1) else $display("p1 check fail");
       ...
a4: assert property(p4) else $display("p4 check fail");
a5: assert property(p5) else $display("p5 check fail");
```

- Coverage Measurement

Since functional coverage is not yet supported, we were not able to compare SystemVerilog and Vera in this aspect. However we found it interesting to try assertion coverage which is not provided by Vera.
We used the same properties to evaluate this feature. The **cover** statement enforces a property as a coverage specification.

```
c1: cover property(p1);
       ...
c4: cover property(p4);
c5: cover property(p5);
```

At the end of simulation, there is a coverage report for each property which displays the number of times the property was attempted, the number of times the property succeeded, and the number of times the property failed.

5.2 Metrics

The whole testbench and test stimulus were developed first on Vera language and then on SystemVerilog. The time of test execution was measured to compare the performance. The metrics that were collected are presented in Table 3 and Figure 7 below:

Table 3. SystemVerilog vs. Vera Code Size in LOC

Code Size metrics	SystemVerilog	Vera
Environment (drivers + responder)	420 LOC	588 LOC
Stimulus	209 LOC	274 LOC

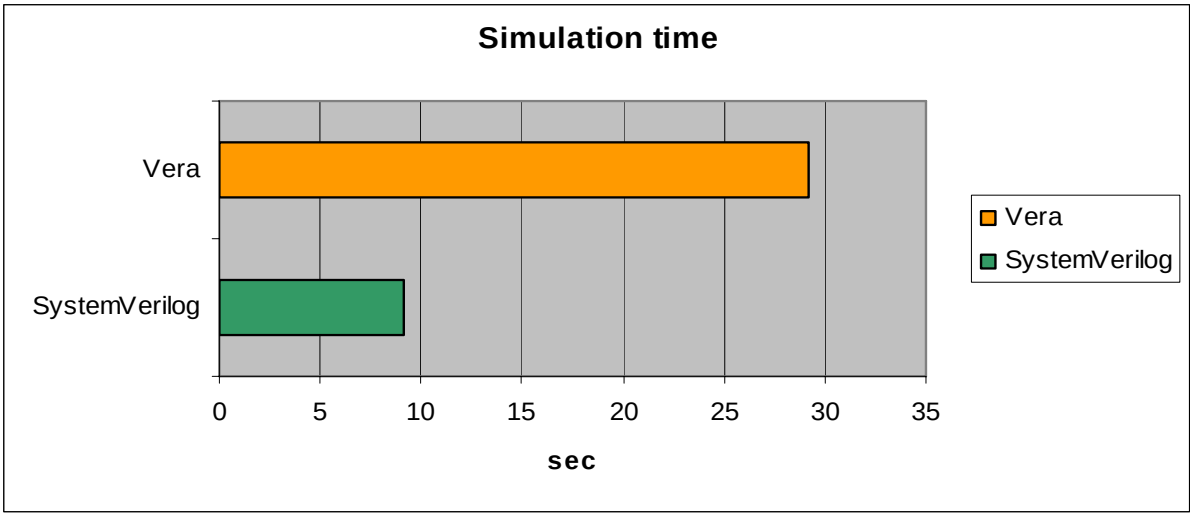

Figure 7. SystemVerilog vs. Vera Performance

Having approximately the same code size for the test environment and stimulus, there is a significant increase in code performance for SystemVerilog. Simulation time is about 3 times smaller.

6.0 Conclusions and Recommendations

The main message of our evaluation is that the current SystemVerilog implementation in VCS does not allow using the theoretical power of the language for verification purposes, as the effort put in SystemVerilog support by Synopsys went to the modeling subset of the language. The only exception is the SystemVerilog assertions which can be used as a stand-alone verification means for protocol checking. Such important verification means as separate program block, classes, OOP and functional coverage are currently omitted.

At the same time, many EDA vendors claim support of SystemVerilog in their products. However there is no full support of the language as specified by Accellera in any existing tool. Vendors implement a subset of the language, which varies from one product to another. It obviously puts significant limitation on portability of the designs, testbenches and tests and slows down SystemVerilog adoption in semiconductor manufacturers' designs and test environments. The feedback from EDA vendors for summer 2004 was like:

- "Full verification support is expected about mid-end of 2005" (Mentor Graphics, Cadence)

- "Use Native Testbench if you are looking for performance, use Vera if you need stable solution" (Synopsys)

Nevertheless the potential of using the language in verification is huge because of several reasons:

- it is an industry standard,
- simulation is significantly faster (as was proven by our evaluation, even though we used a small subset of dedicated to verification constructs),
- it incorporates support for modern verification technologies, such as transaction-based verification, coverage driven verification, assertion based verification, reference verification methodology, which give freedom of choice to verification engineers.

7.0 References

1. SystemVerilog 3.1 Accellera's Extensions to Verilog-2001
2. Synopsys VCS 7.2 User's Guide
3. http://www.systemverilog.org
4. http://www.accellera.org
5. http://www.systemverilognow.com (online seminar)
6. http://www.snug-universal.org/papers/papers.htm